Time Complexity

Reading

Section 7.1

Practice problems (page 294): 7.1, 7.2

Challenge: 7.47

Time Complexity

In this and subsequent sections we discuss problems related to the resources needed to solve a problem, rather than whether the problem is solvable or not. The two main resources at our disposal are time and space. We will mostly focus on time.

> Let M be a deterministic Turing Machine that halts on all inputs. The **time complexity** or **running time** of M is the function $f\colon \mathbb{N} \to \mathbb{N}$ such that for each natural number n we define $f(n)$ to be the maximum number of steps that M takes on any input of length n.
>
> We also say that M runs on time $f(n)$, and that M is a $f(n)$-time Turing Machine.

What we are mostly interested in is how this function grows with n. This is usually the topic of a field called **asymptotic analysis**. It revolves, amongst other concepts, around the two key definitions of **big-O** and **little-o** notation.:

For example we could imagine two Turing Machines performing the same task. One makes one pass through its input and has running time n, the other does some initial state transitioning before passing through the input, for a running time of $n+5$. Clearly that "plus 5" is irrelevant when the input has any moderate size n.

> For two functions $f, g\colon \mathbb{N} \to \mathbb{R}^+$ we write:
>
> $$f(n) = O(g(n))$$
>
> if there are positive integers c and n_0 so that for all $n \geq n_0$ we have:
>
> $$f(n) \leq cg(n)$$
>
> We also say that $g(n)$ is an **(asymptotic) upper bound** for $f(n)$.

For instance $2n = O(n)$, but n^2 is not $O(n)$. In general a polynomial of degree k would be $O(n^k)$, as lower terms are relatively insignificant.

The big-O notation expresses the idea that a function is asymptotically no more than another. A related notation, little-o, is used to express the idea that a function is asymptotically less than another:

For two functions $f, g\colon \mathbb{N} \to \mathbb{R}^+$ we write:

$$f(n) = o(g(n))$$

if for every positive $c > 0$ there is an n_0 such that for all $n \geq n_0$ we have:

$$f(n) \leq cg(n)$$

For those familiar with the limit notation, this is equivalent to:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Some examples: $n = o(n^2)$, $n \log n = o(n^2)$, $n = o(n \log n)$.

Some algebra for the big-O notation will be useful:

- If $f(n) = O(g(n))$ and $h(n) = O(f(n))$, then $h(n) = O(f(n))$ (i.e. $O(O(f(n))) = O(f(n))$. We can "compose" big-O notation.
- If $f(n) = O(g(n))$, then $cf(n) = O(g(n))$ (i.e. $O(cg(n)) = O(g(n))$). We can ignore multiplicative constants.
- If $f(n) = O(g(n))$, and $h(n) = O(f(n) + g(n))$, then $h(n) = O(g(n))$. We can ignore additive powers that are at most equal to the term they are being added to.
- If $f(n) = O(g(n))$, then $h(n)f(n) = O(h(n)g(n))$ (i.e. $h(n)O(g(n)) = O(h(n)g(n))$).

The time complexity of problems allows us to define the notion of a *time compexity class*:

> If $t\colon \mathbb{N} \to \mathbb{R}^+$, the **time complexity class** TIME$(t(n))$ is the collection of all languages that are decidable by an $O(t(n))$ Turing Machine.
>
> Note that it is not required that all TMs that decide the language have that running time limit, but only that at least one such TM does.

The time complexity classes form a tower: For instance every language in TM(n) is also in TM$(n \log n)$, and every language in TM$(n \log n)$ is in turn in TM(n^2) and so on. "Smaller" time complexity classes correspond to stronger restrictions on the running time of the TM.

We are typically interested in finding, for a given language/problem, the appropriate time complexity class. For instance if a language is shown to be in TM(n^2), we would be interested in seeing if it is also in a "smaller" class like TM$(n \log n)$ or TM(n), or whether that n^2 is "the best we can do".

In most Algorithms classes, these ideas are explored further.

An example

Let us consider an example to explore these ideas further, with the language $A = \{0^k 1^k \mid k \geq 0\}$.

Here is a first attempt at a Turing Machine that decides this problem:

M_1: On input string w:

a. Scan the tape and reject if a 0 is found to the right of a 1.
b. Repeat if both 0s and 1s remain on the tape:
    1. Scan across the tape, crossing off a single 0 and a single 1.
c. If 0s or 1s remain after all pairs have been crossed off, reject. Otherwise accept.

Now we need to determine the running time of this TM.

- Part a takes $n = O(n)$ steps, as it requires a single pass through the input.
- Each time Part b1 is performed, it requires a pass through at least half the input, for $n/2 = O(n)$ steps.
- This will be repeated for $n/2 = O(n)$ times, effectively once for each 0 in the list. So the total running time for this part b is $O(n)O(n) = O(n^2)$.
- Part c requires a single pass through the list, for another $O(n)$ steps.
- So the total running time would be $O(n) + O(n^2) + O(n) = O(n^2)$. The running time is completely dominated by the expensive second step.

So this tells us that the language A is in TIME(n^2).

The question now is whether there is a faster TM for the same problem. In fact, there is:

M_2: On input string w:

a. Scan the tape and reject if a 0 is found to the right of a 1.
b. Repeat as long as some 0s and some 1s remain on the tape:
    1. Scan across the tape, ensuring that the parity of the remaining number of 0s and 1s is the same (i.e. both odd or both even). If it is not, reject
    2. Make another pass through the tape, crossing every other 0 starting from the first one, and similarly crossing every other 1 starting from the first one.
c. If 0s or 1s remain after all pairs have been crossed off, reject. Otherwise accept.

This turns out to be much faster, for a running time of $O(n \log n)$. The key part is an estimate of the number of steps needed to perform part b. Every pass through the steps in b reduces the number of viable numbers by half. So step b is going to be repeated for a number of times k so that $n \approx 2^k$, in other words $k = O(\log n)$. Since each pass requires time $O(n)$, the overall time it takes to get through the second phase is $O(n \log n)$. So the time complexity for the whole TM is $O(n)+O(n \log n)+O(n) = O(n \log n)$.

In other words, we actually have shown that the language A is in TIME$(n \log n)$. In fact it turns out that this result cannot be improved further.

Interestingly, adding more tapes significantly increases our power: In this case this problem can be easily solved in time $O(n)$ using a 2-tape machine.

Exercise: Show this. Design a $O(n)$-time 2-tape TM that decides the language A.

This is an important difference between complexity theory and computability theory: Questions of time complexity depend on the computational model used, while questions of decidability and so on did not.

Complexity Relationships between models

In this section we determine how the different models relate in terms of complexity. There are two main cases to consider, the multi-tape model and the non-deterministic model.

Multi-tape model and complexity

> If $t(n)$ is a function where $t(n) \geq n$. Then every $t(n)$-time multitape Turing Machine has an equivalent $O(t^2(n))$-time single-tape Turing Machine.

Note here that the number k of tapes is fixed and independent of the input size n, so it is treated simply as a constant.

The assumption that $t(n) \geq n$ is not that restrictive: Almost any TM would need to at least read its input once, and that takes n steps.

The proof of this theorem is not all that surprising. If we suppose that we have a k-tape TM and we are processing input of size n, then recall how we thought of this as a single-tape machine:

- We simulated the multi-tape contents in one long steam. Each tape can have at most $t(n)$ contents in it (as each step can increase the content at most by one slot). So the overall contents would fit into $O(t(n))$ space in the tape.
- To determine a next move, the TM will have to first read all these contents to find the state of the corresponding multi-tape machine. This takes $O(t(n))$ time.
- After it determines what to do, it must similarly go back and update the tapes, which also takes $O(t(n))$ time.
- So each step through this simulation takes us $O(t(n))$ time to perform. We need to perform $O(t(n))$ such steps, for a total running time of $O(t^2(n))$.

Non-deterministic model and complexity

> If N is a non-deterministic Turing Machine that is a decider, then the **running time** of N is a function $f: \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

So if the TM is fed input of size n, then every branch of the non-deterministic computation must end it time $f(n)$.

> Suppose $t(n) \geq n$ is a function. Then every $t(n)$-time non-deterministic single-tape Turing Machine has an equivalent $2^{O(t(n))}$-time deterministic single-tape Turing Machine.

The idea of the proof is similar. We need to estimate the running time of the deterministic TM that simulates the corresponding non-deterministic TM.

Recall that the possibilities of the TM can be thought of as a tree, where each node has at most b branches, and the depth is bounded by $t(n)$. So there is a total of $O(b^{t(n)})$ nodes/leaves to consider. We need to figure out the time it takes for each one of those to be examined. We know in fact that time is $O(t(n))$.

So the total running time is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. Now this is the running time of the 3-tape TM that simulates the non-deterministic TM. We can in turn simulate that TM with a single-tape machine at time $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

You will notice that the corresponding single-tape TM is considerably slower than its non-deterministic counterpart. This should not be all that surprising. It will be a crucial component in future discussions.